Exploration and Implementation of Neural Ordinary Differential Equations

Long Nguyen, Andy Malinsky
Department of Computer Science
Mathematics
Arcadia University
Glenside, Pennsylvania 19038 USA

April 21, 2020

Abstract

Neural ordinary differential equations (ODEs) have recently emerged as a novel approach to deep learning, leveraging the knowledge of two previously separate domains, neural networks and differential equations. In this paper, we first examine the background and lay the foundation for traditional artificial neural networks. We then present neural ODEs from a rigorous mathematical perspective, and explore their advantages and trade-offs compared to traditional neural nets.

Contents

1 Introduction

Deep learning is a subset of machine learning, which in turn is a subset of artificial intelligence. To automate a computer's learning process, this class of algorithms and numerical methods utilize artificial neural network architectures, loosely inspired by biological neural networks found in the human brain. Recent astonishing successes in accurate predictive modeling and self-teaching algorithms have been attributed to a combination of mathematical knowledge, computer engineering advances, and most importantly the unprecedented availability of training data in the modern era. The magic behind the scenes, however, is a simple framework rooted in matrix and vector multiplication, as well as differentiation rules [8]. Therefore, the extension of neural networks into the differential equations field in recent years have really opened up new and exciting areas of research for deep learning. In this paper, we address the use of ordinary differential equations implemented with deep neural networks to further improve training efficiencies.

Section 2 introduces and explains the essential concepts and mathematics behind artificial neural networks, together with a simple classification example. General notations are established to facilitate discussions about neural networks, and later on, neural ODEs. Then, we present the gradient descent method and its proof to finally generalize a back propagation algorithm. Section 3 extends the conception of neural networks with differential equations, showing the motivation of neural ODEs from the use of residual neural networks. Similar to traditional nets, we introduce and examine a back propagation algorithm via the adjoint method.

In addition, each section includes a programming implementation in the Julia language to demonstrate neural networks and neural ODEs in action. While many robust and elegant machine learning libraries are available, such as the Julia-native DiffEqFlux.jl [13] package, we attempt to build our framework from scratch as much as possible in order to augment our understanding of neural nets, and also to provide a guide for others interested in what goes on under the hood. All the codes in this paper are attached in the Appendix, and can be accessed from this Github repository.

2 Artificial Neural Networks

This section introduces and explains the notations and mathematical foundation of artificial neural networks.

2.1 Example of a Neural Network

In essence, neural networks are valuable tools to make future predictions based on currently available data. The architecture is applicable to many different family of problems, but we choose a classification one to demonstrate its ability of extrapolating hidden dynamics purely from data. Two main approaches to the classification problems in machine learning are supervised and unsupervised algorithms: while supervised learning attempts to approximate

the mapping of inputs to desired, labeled outputs, the unsupervised approach learns to group data point without predetermined categories. For example, say we have a collection of images of animals, and aim to develop a computer model to automatically distinguish the different species. A supervised network is provided with the species label of each image to govern the learning direction and improve its accuracy. However, an unsupervised method will be responsible for generating its own classes and categories by clustering or association algorithms.

To keep our example simple without sacrificing generality, we choose a relatively straightforward binary classification problem. We first define an arbitrary equation as the classifying model, here x^3-3x+2 in Figure 1, to split the $\mathbb{R}^2$ space into two groups, where points above or on the model are labeled 1, while others are labeled 0. The objective is for a neural network to learn this hidden model with a randomly generated collection of data points, together with their corresponding labels. As such, this example utilizes the supervised approach to machine learning.

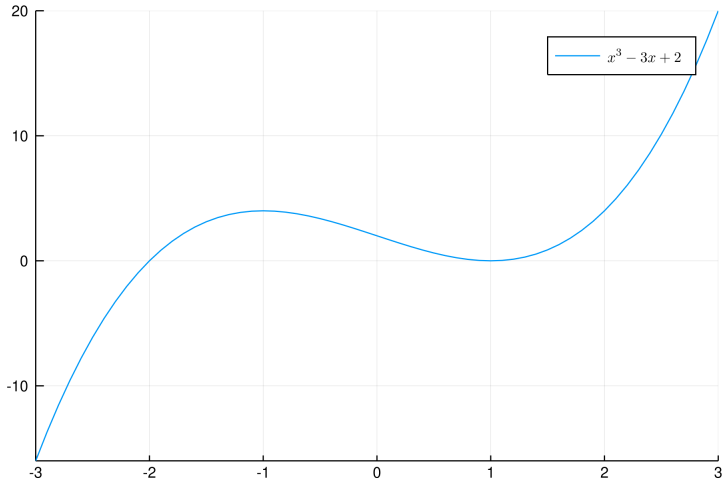


Figure 1: An arbitrary equation, here $x^3 - 3x + 2$, is chosen as the true classifying model. This is the function we want our neural network to ultimately capture.

Given this model, we generate a collection of data points in the space and label them accordingly. As the neural network only has access to this data set in training, we demonstrate its ability to approximate hidden dynamics without the need for explicit models.

Represented and depicted by neural circuits, biological neural networks are thought of as an interconnected system of neurons, where chemical and electrical signals traverse to activate different functions of the human brain. An artificial neural network, inspired by this system, is constructed in a similar manner as multiple *layers* of neurons. Figure 3, for instance, displays an artificial neural network with four layers. The first layer, namely the *input layer*, is where the input data enters the network. The last layer is known as the *output layer*, storing the outputs produced by the network. Every layer in between is known as a *hidden layer*, where mathematical computations transform the inputs into the outputs.

In the simplest form, each neuron in layer l are only connected to each neuron in layers $l-1$ and $l+1$. At each hidden layer, the output from the previous layer is taken as input, and

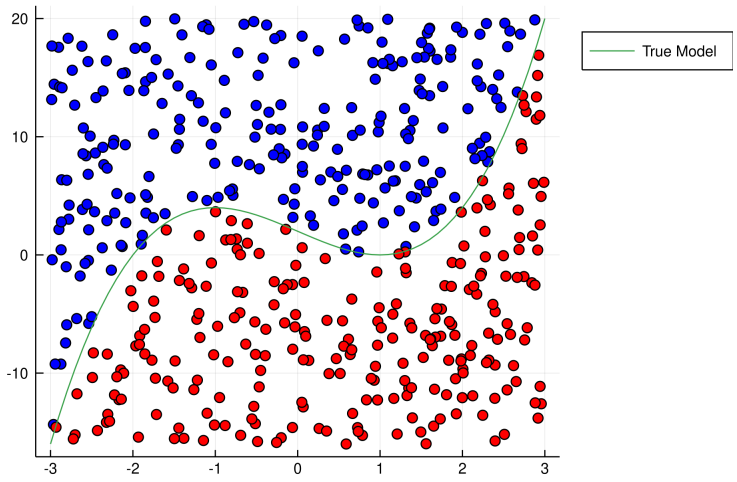


Figure 2: Labeled data points. Blue points in the graph lie on or above the model equation, and so are labeled 1. Red points are labeled 0 and lie below the model equation.

in turn its output is the input of the next layer. Each connection between a pair of neurons carries a *weight* value, which is what we want to optimize during the training process. Every neuron outputs a single real number, and this real number is then multiplied by the corresponding weight value before an additional predetermined and constant *bias* value is added. From a given layer l, we can represent all the real number outputs as a vector a, all the weights as a matrix W, and all the constant biases as a vector b. Thus, let x be the vector of outputs from a layer l, then it is calculated using the formula $x = Wa + b$. An *activation function*, usually one that maps to the range from 0 to 1, is applied to each output before x is sent to the next layer. An example of a commonly used nonlinear activation function is the sigmoid function, defined by $\sigma(x)$:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

It is important to note that at layer l the number of columns in W matches the length of vector a from the previous layer $l - 1$, and the number of rows matches the length of vector a at the current layer l. The length of vector b also matches the number of neurons at the current layer l.

Taking Figure 3 as an example, we can apply our mathematical notations, as adapted from [8], to help us denote the computations at each layer. Since the input layer has two neurons, the input data is a vector $a \in \mathbb{R}^2$. We have established the corresponding dimensions of the weights and biases in the next layer above, they will be represented as a matrix $W^{[2]} \in \mathbb{R}^{2x2}$ and a vector $b^{[2]} \in \mathbb{R}^2$, respectively. The resulting output from layer two therefore is computed as

$$\sigma(W^{[2]}a + b^{[2]}) \in \mathbb{R}^2.$$

At this point, the output from layer two will serve as input to layer three. Since the third layer has three neurons, and the received input is in $\mathbb{R}^2$, we respectively represent the weights and biases in layer three as a matrix $W^{[3]} \in \mathbb{R}^{3x2}$ and a vector $b^{[3]} \in \mathbb{R}^3$. The output

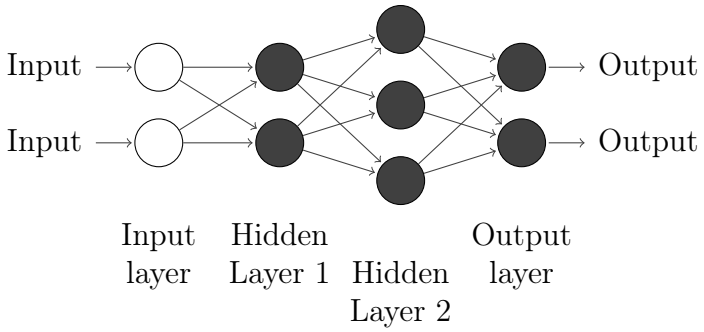


Figure 3: A neural network with four layers [8].

of layer three is then

$$\sigma(W^{[3]}\sigma(W^{[2]}a + b^{[2]}) + b^{[3]}) \in \mathbb{R}^3.$$

Finally, the output from layer three is taken as input to layer four, the output layer. Here, the output layer has only two neurons, and the received input is in $\mathbb{R}^3$. Consequently, the weights and biases are a matrix $W^{[4]} \in \mathbb{R}^{2x3}$ and a vector $b^{[4]} \in \mathbb{R}^2$, respectively. The resulting output from layer four, and the entire neural network in general, is thus

$$F(x) = \sigma(W^{[4]}\sigma(W^{[3]}\sigma(W^{[2]}a + b^{[2]}) + b^{[3]}) + b^{[4]}) \in \mathbb{R}^2. \tag{2}$$

Equation (2) captures the underlying mapping of our neural network. The goal of training is to pick and choose the appropriate weights and biases, or *optimize the parameters*, such that the output can be used to maximize some objectives or to minimize some costs. In the context of our binary classifier example, we would like $F(x)$ to be able to predict the actual labels as as accurately as possible. A simple scheme is to assign the label 1 for any non-negative value of $F(x)$, and the label 0 otherwise. As such, given any data point $x \in \mathbb{R}^2$, we could run its coordinates through the neural network and assign the appropriate label.

In order to evaluate the performance of our neural network and to adjust training trajectories accordingly, we require a *cost function*. Essentially, the role of the cost function is to measure the difference between the obtained output and the desired one. To achieve a reasonable classification accuracy would most likely require more than just one iteration through the data set. At each iteration, we will adjust the weights and biases in such a way that minimizes the cost function. As such, training a neural network is an optimization problem where we hope to find a global minimum, or a local minimum close enough. Figure 4, for instance, plots the results of our simple neural network after training against the true model. We observe that while not perfect, our neural network performs fairly well, especially when we only use one hidden layer in training.

In the next section, we will generalize the neural network setup for any arbitrary number of layers.

2.2 The General Set-up

Let us revisit the point that a neural network model consists of multiple layers to transform the input into the output. A general layer consists of a number of neurons, receiving from

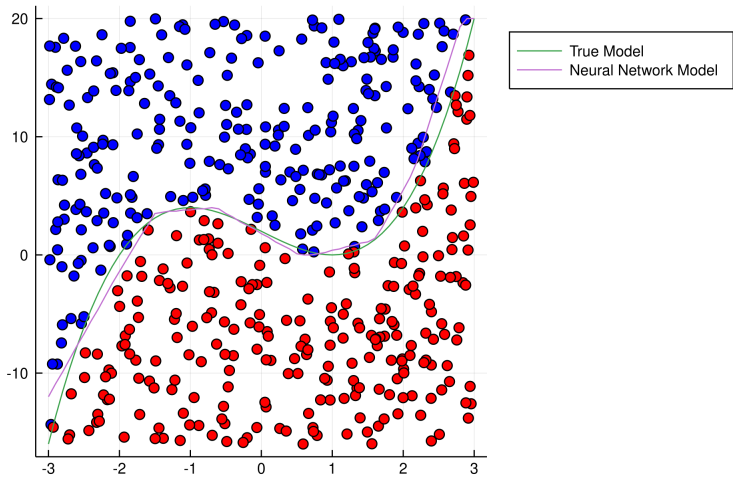


Figure 4: Visualization of prediction output from an artificial neural network applied to the data in Figure 2.

each neuron of the previous layer and contributing to every neuron of the next. As inspired by biological neural networks, an activation function is applied to the neurons of each layer before their real values are passed to the next. The first layer is the input to the model, and the last is the output produced. In between are *hidden layers*, the intermediary computations of the neural network.

We now introduce mathematical notations for neural networks that will be used for the remaining of this paper. The notations are consistent with C. Higham and D. Higham's paper [8], and aim to formulate and define neural networks using mathematical conventions.

A generalized neural network contains L layers, each layer l in turns contains n_l neurons. Thus, the network takes in an input from $\mathbb{R}^{n_1}$ and return an output in $\mathbb{R}^{n_l}$. In addition, the weights and biases at layer l shall be denoted by $\mathbf{W}^{[l]} \in \mathbb{R}^{n_l} \times \mathbb{R}^{n_l - 1}$ and $\mathbf{b}^{[l]} \in \mathbb{R}^{n_l}$, respectively. We use $w_{jk}^{[l]}$ to denote a specific weight applied to the neuron k of layer $l-1$ to produce the neuron j of layer l. By the same token, $b_j^{[l]}$ denotes the bias of the neuron j at layer l.

Figure 5 visualizes a neural network with $L = 5$ layers. Using the introduced notations, the number of neurons at each layer is $n_1 = 4$, $n_2 = 3$, $n_3 = 4$, $n_4 = 5$, and $n_5 = 2$. The weights are $\mathbf{W}[2] \in \mathbb{R}^{3\times4}$, $\mathbf{W}[3] \in \mathbb{R}^{4\times3}$, $\mathbf{W}^{[4]} \in \mathbb{R}^{5\times4}$, $\mathbf{W}^{[5]} \in \mathbb{R}^{2\times5}$; and the biases are $\mathbf{b}^{[2]} \in \mathbb{R}^3$, $\mathbf{b}^{[3]} \in \mathbb{R}^4$, $\mathbf{b}^{[4]} \in \mathbb{R}^5$, $\mathbf{b}^{[5]} \in \mathbb{R}^2$.

We further denote $z_j^{[l]}$ to be the intermediate output for neuron j at layer l, before the activation function is applied. We then denote $a_j^{[l]}$ to be the output of neuron j at layer l, after the activation function is applied. We can then formulate the algorithm for a generalized neural network as follows:

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \tag{3}$$

$$\mathbf{a}^{[1]} = x \in \mathbb{R}^{n_1} \tag{4}$$

$$\mathbf{a}^{[l]} = \sigma(\mathbf{z}^{[l]}), for l = 2, 3, \ldots, L \tag{5}$$

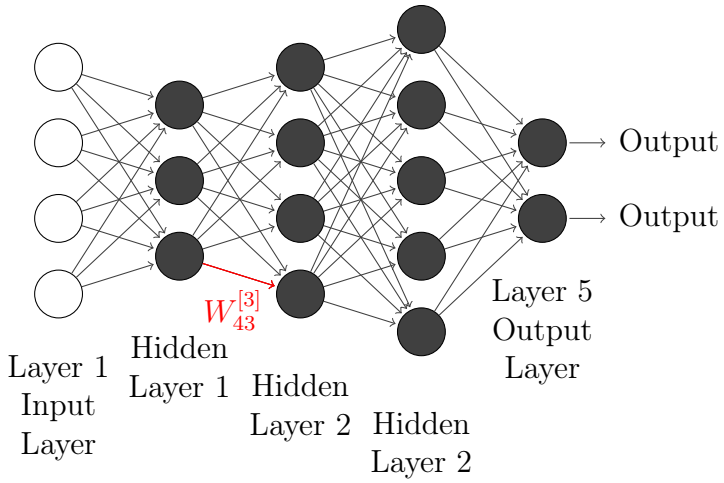


Figure 5: A neural network with five layers. The highlighted path denotes the weight applied to neuron 3 of layer 2 and then fed to neuron 4 in layer 3.

To train the neural network, training data need to be fed into the model and evaluated. Suppose the training set consists of N inputs in $\mathbb{R}^{n_1}$, $\{\mathbf{x}^{\{i\}}\}_{i=1}^N$, and the corresponding outputs in $\mathbb{R}^{n_L}$, $\{\mathbf{y}^{\{i\}}\}_{i=1}^N$. Feeding the inputs to the model will produce the corresponding predictions, $\mathbf{a}^{[L]}(\mathbf{x}^{\{i\}})$. In order to assess the accuracy of the neural network for training and validation, a cost objective to measure the error between the predictions and the observed outputs is generalized as

$$\text{Cost} = \frac{1}{N}\sum_{i=1}^{N} c(\mathbf{y}(\mathbf{x}^{\{i\}}) - \mathbf{a}^{[L]}(\mathbf{x}^{\{i\}})), \tag{6}$$

where c is the chosen cost function. In the case of quadratic cost, the objective to be minimized is

$$\text{Cost} = \frac{1}{N}\sum_{i=1}^{N} \frac{1}{2}||(\mathbf{y}(\mathbf{x}^{\{i\}}) - \mathbf{a}^{[L]}(\mathbf{x}^{\{i\}}))||_2^2. \tag{7}$$

The factor $\frac{1}{2}$ is included for simplification purposes later on with differentiation. In the next section, we will explore how to optimize the cost objective for neural networks.

2.3 Gradient Descent

In general, deep learning using neural networks aim to find the optimal weights and biases that minimize the cost objective. To simplify the notations, we imagine they are flattened into a single parameter vector $\mathbf{p}$. The cost objective can then be characterized as a function of s parameters, mapping from $\mathbb{R}^s$ to $\mathbb{R}$.

Traditionally, this optimization problem can be solved by taking the partial derivative of the cost objective with respect to each parameter, and setting the partial derivatives to 0 to formulate a system of equations. However, considering that neural networks typically utilize a large number of parameters, it is not computationally efficient to solve for the optimal solution.

Instead, the method of choice in deep learning is *gradient descent*, where we generate a sequence of vectors in $\mathbb{R}^s$ to iteratively approximate the optimal solution that minimizes the cost function. Suppose the initial parameter vector is $\mathbf{p}$, our goal is to find the appropriate change $\Delta\mathbf{p}$ to compute the next iteration. We recall that the cost objective can be expanded using the Taylor series [14]:

$$\text{Cost}(\mathbf{p}+\Delta\mathbf{p}) = \text{Cost}(\mathbf{p}) + \sum_{i=1}^{s} \frac{\partial \text{Cost}(\mathbf{p})}{\partial p_i}\Delta p_i + \frac{1}{2}\sum_{i=1}^{s}\sum_{j=1}^{s} \frac{\partial^2 Cost(\mathbf{p})}{\partial p_i \partial p_j}\Delta p_i \Delta p_j + ... \tag{8}$$

Assume $\Delta\mathbf{p}$ is small enough, we arrive at the approximation

$$\text{Cost}(\mathbf{p}+\Delta\mathbf{p}) \approx \text{Cost}(\mathbf{p}) + \sum_{i=1}^{s} \frac{\partial \text{Cost}(\mathbf{p})}{\partial p_i}\Delta p_i. \tag{9}$$

Denoting the *gradient* vector as $\nabla\text{Cost}(\mathbf{p})$, where $\nabla\text{Cost}(\mathbf{p})_i$ is the partial derivative of the cost objective with respect to p_i, gives us the more concise form

$$\text{Cost}(\mathbf{p}+\Delta\mathbf{p}) \approx \text{Cost}(\mathbf{p}) + \nabla\text{Cost}(\mathbf{p})^T\Delta\mathbf{p}. \tag{10}$$

To minimize the cost objective then, we just need to choose $\Delta\mathbf{p}$ so that $\nabla\text{Cost}(\mathbf{p})^T\Delta\mathbf{p}$ is the most negative at each iteration. Notice the following inequality from the dot product

$$\nabla\text{Cost}(\mathbf{p})^T\Delta\mathbf{p} = ||\nabla\text{Cost}(\mathbf{p})||_2||\Delta\mathbf{p}||_2 cos(\nabla\text{Cost}(\mathbf{p}), \Delta\mathbf{p}) \tag{11}$$

$$\geq -||\nabla\text{Cost}(\mathbf{p})||_2||\Delta\mathbf{p}||_2. \tag{12}$$

The most negative descent is achieved if the cosine of the angle between $\nabla\text{Cost}(\mathbf{p})$ and $\Delta\mathbf{p}$ is -1, in other words, if $\Delta\mathbf{p}$ is in the opposite direction of $\nabla\text{Cost}(\mathbf{p})$. Considering that the approximation holds for small $\Delta\mathbf{p}$, we define a small step size η and make the update

$$\mathbf{p} \to \mathbf{p} - \eta\nabla\text{Cost}(\mathbf{p}) \tag{13}$$

.

The step size η is more commonly referred to as the *learning rate*, and equation (13) formulates the gradient descent algorithm for neural network optimization. We start with the initial parameters $\mathbf{p}$ and iteratively update $\mathbf{p}$ until a stopping criterion is satisfied or the maximum number of iterations is achieved.

Even then, computing the cost over the entire data set could still be computationally expensive. One alternative approach is to randomly select a training point, and only evaluate the cost at such point. This is referred to as stochastic gradient descent, the scheme is as follows:

1. Uniformly choose a random training point i from the set 1,2,...,N
2. Update

$$\mathbf{p} \to \mathbf{p} - \eta\nabla\text{Cost}_{x_i}(\mathbf{p}). \tag{14}$$

One thing to keep in mind, however, is that the above algorithm chooses training point *with replacement*. That is, it is equally likely for a training point i to be picked each iteration. Alternatively, we could randomize the data set and train our model through each training point to complete an *epoch*:

1. Shuffle the data set
2. Update
$$\mathbf{p} \to \mathbf{p} - \eta \nabla \text{Cost}_{x_i}(\mathbf{p}). \tag{15}$$

While stochastic gradient descent is computationally inexpensive, updating the neural network with random data points can be unstable. One other popular approach is to use mini-batching, where we split the data set into smaller batches. This has the benefit of better stability with more data points per run, while relatively efficient compared to training on the entire data set

1. Split the data set into smaller mini-batches $X_1, X_2, ..., X_m$.
2. Update
$$\mathbf{p} \to \mathbf{p} - \eta \nabla \text{Cost}_{X_i}(\mathbf{p}). \tag{16}$$

2.4 Back Propagation

Now we are ready to move back from our generalized parameter vector $\mathbf{p}$ to the weight martices $w_{jk}^{[l]}$ and bias vectors $b_j^{[l]}$. Since we want to minimize the cost function, our job is to compute the partial derivatives with respect to the weight matrices and bias vectors. These partial derivatives will tell us at each iteration how much each variable impacts the cost function. Focusing on a fixed training point, the cost function from (7) can be simplified in terms of just weights and biases,

$$\text{Cost} = \frac{1}{2}||(y - a^{[L]})||_2^2 \tag{17}$$

We now define $\delta^{[l]} \in \mathbb{R}^{n_l}$ as

$$\delta_j^{[l]} = \frac{\partial \text{Cost}}{\partial z_j^{[l]}}, \text{ for } 1 \leq j \leq n_l \text{ and } 2 \leq l \leq L \tag{18}$$

We let the expression for $\delta_j^{[l]}$ correspond to the error, or sensitivity to the cost function, in neuron j at layer l. Let us also define the Hadamard product of two vectors. Given two vectors $x, y \in \mathbb{R}^n$, the Hadamard product $(x \circ y)_i = x_i y_i$, where $x \circ y \in \mathbb{R}^n$. As such, the Hadamard product results in the pairwise multiplication of the components of two vectors. The lemma below lists the properties of the backpropagation algorithm. We use the chain rule to derive the partial derivatives and to prove each equation of the lemma.

Lemma 2.1.

$$\delta^{[L]} = \sigma'(z^{[L]}) \circ \frac{\nabla \text{Cost}}{a^{[L]}} \tag{19}$$

$$\delta^{[l]} = \sigma'(z^{[l]}) \circ (W^{[l+1]})^T \delta^{[l+1]} \qquad \text{for } 2 \leq l \leq L-1 \tag{20}$$

$$\frac{\partial \text{Cost}}{\partial b_j^{[l]}} = \delta_j^{[l]} \qquad \text{for } 2 \leq l \leq L-1 \tag{21}$$

$$\frac{\partial \text{Cost}}{\partial w_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]} \qquad \text{for } 2 \leq l \leq L-1 \tag{22}$$

$$\tag{23}$$

Proof. We begin by proving (19). We know from (5) that $a^{[L]} = \sigma(z^{[L]})$ for $l = L$, so we can derive the following:

$$\frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = \sigma'(z)$$

Then, taking the partial derivative of the Cost function,

$$\frac{\partial \text{Cost}}{\partial a_j^{[L]}} = -(y_j - a_j^{[L]}) = a_j^{[L]} - y_j.$$

Finally, by chain rule starting with our definition of $\delta_j^{[L]}$,

$$\sigma_j^{[L]} = \frac{\partial \text{Cost}}{\partial z_j^{[L]}} = \frac{\partial \text{Cost}}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = (a_j^{[L]} - y_j)\sigma'(z_j^{[L]}) \tag{24}$$

.

Next, we show (20). By definition, at layer $l+1$,

$$z^{[l+1]} = W^{[l+1]} a^{[l]} + b^{[l+1]}.$$

For an arbitrary neuron i at layer $l+1$, it follows that

$$z_i^{[l+1]} = \sum_{j=1}^{n_l} W_{ij}^{[l+1]} a_j^{[l]} + b_i^{[l+1]}.$$

The partial derivative of $z_i^{[l+1]}$ with respect to a neuron j at layer l is then

$$\frac{\partial z_i^{[l+1]}}{\partial a_j^{[l]}} = W_{ij}^{[l+1]}.$$

Notice that $\frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} = \sigma'(z_j^{[l]})$, using the chain rule gives us

$$\begin{aligned} \frac{\partial \partial z_i^{[l+1]}}{\partial z_j^{[l]}} &= \frac{\partial z_i^{[l+1]}}{\partial a_j^{[l]}} \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \\ &= W_{ij}^{[l+1]} \sigma'(z_j^{[l]}). \end{aligned}$$

We are then able to compute the partial derivative of the Cost with respect to neuron j at layer l by propagating through its contribution to each of the neurons at layer $l+1$:

$$\begin{aligned}\frac{\partial \text{Cost}}{\partial z_j^{[l]}} &= \sum_{i=1}^{n_{l+1}} \frac{\partial \text{Cost}}{\partial z_i^{[l+1]}} \frac{\partial \partial z_i^{[l+1]}}{\partial z_j^{[l]}} \\ &= \sum_{i=1}^{n_{l+1}} \delta_i^{[l+1]} W_{ij}^{[l+1]} \sigma'(z_j^{[l]}),\end{aligned}$$

which is the component-wise form of equation (20).

Now, to prove (21) we start with out definition of $z_j^{[l]}$ from (3) at a component j,

$$z_j^{[l]} = W^{[l]} a_j^{[l-1]} + b_j^{[l]}$$

Taking the partial derivative with respect to the bias vector we get,

$$\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1$$

Then, by the chain rule and our definition of $\delta_j^{[l]}$,

$$\frac{\partial \text{Cost}}{\partial b_j^{[l]}} = \frac{\partial \text{Cost}}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = \frac{\partial Cost}{\partial z_j^{[l]}}(1) = \delta_j^{[l]} \tag{25}$$

Lastly, to show (22) we begin with the component wise version of $z_j^{[l]}$,

$$z_j^{[l]} = \sum_{k=1}^{n_{l-1}} w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}$$

Taking the partial derivative,

$$\frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = a_k^{[l-1]}$$

Then, by chain rule and definition of $\delta_j^{[l]}$,

$$\frac{\partial Cost}{\partial w_{jk}^{[l]}} = \frac{\partial Cost}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial \partial w_{jk}^{[l]}} = \frac{\partial Cost}{\partial \partial w_{jk}^{[l]}}(a_k^{[l-1]}) = \delta_j^{[l]} a_k^{[l-1]} \tag{26}$$

□

2.5 Neural Network Implementation in Julia

The full Julia implementation can be found in Appendix A1. We include the entire construction of a generic neural network from scratch without any external libraries, taking advantage of Julia's mathematical and scientific computing capabilities. The code to produce the classifier example used in this section is in Appendix A2, where we implement a neural network of one hidden layer and train the model to approximate the classification function.

3 Neural Ordinary Differential Equations

3.1 Residual Neural Networks

The inspiration for using differential equations within a neural network framework stems from the recent successes of residual neural networks (ResNet), especially within image recognition [7]. While an increase in layers would usually yield an increase in performance, large neural networks could suffer from diminishing gradient: the gradient of the Cost with respect to earlier layers becomes too minuscule to actually update the parameters. Residual networks, however, are able to resolve this issue by allowing shortcuts over some layers. This architecture is structured in such a way that each layer can be defined as a finite transformation:

$$h_{t+1} = h_t + g(h_t, \theta_t) \tag{27}$$

where h_t is the hidden state at layer t, g is a dimension preserving function, and θ is a vector of parameters. Such formulation, however, can also be interpreted as an Euler discretization of a continuous ordinary differential equation [9]. The intuition to treat a continuous differential equation as a series of infinitely small residual steps. Starting with (27) and some constant $\Delta t \in \mathbb{R}$,

$$\begin{aligned} h_{t+1} &= h_t + g(h_t, \theta_t) \\ &= h_t + \frac{\Delta t}{\Delta t} g(h_t, \theta_t) \\ &= h_t + \Delta t f(h_t, \theta_t) \end{aligned}$$

Here, f can be re-interpreted as a step of the Euler's discretization method to approximate a continuous function:

$$\frac{(h_{t+1} - h_t)}{\Delta t} = f(h_t, \theta_t). \tag{28}$$

This inspires us to push the time step Δt to be infinitesimal, and so arrive at a formulation of continuous hidden state dynamics, where each layer h is a discretized evaluation of a continuous function z at time t:

$$\frac{dz(t)}{dt} = f(z(t), \theta(t), t). \tag{29}$$

Figure 6 illustrates this extension by comparing a residual neural network to a continuous one. In this interpretation, the neural network has become an initial value ODE problem, whose solution is the function mapping the input to the output over some time range. In the next section, we further explore this idea of continuous formulation, and introduce the ingredients of a neural ODE.

While still in the early stage of research, Neural ODEs appear to be an exciting alternative worth exploring, with some apparent advantages over standard ResNets. The original paper *Neural Ordinary Differential Equations* [3] received the Best Paper Award at the 2018 Neural Information Processing Systems (NeurIPS) conference. The authors, affiliated with the

Vector Institute at the University of Toronto, tout the parametric and memory efficiency of the model. Furthermore, they are able to draw upon over 100 years of differential equations knowledge and utilize the adaptive computation of modern ODE solver. This means that we can trade longer computation time for better accuracy as necessary. We shall investigate the benefits and potential pitfalls of the neural ODE framework in later sections.

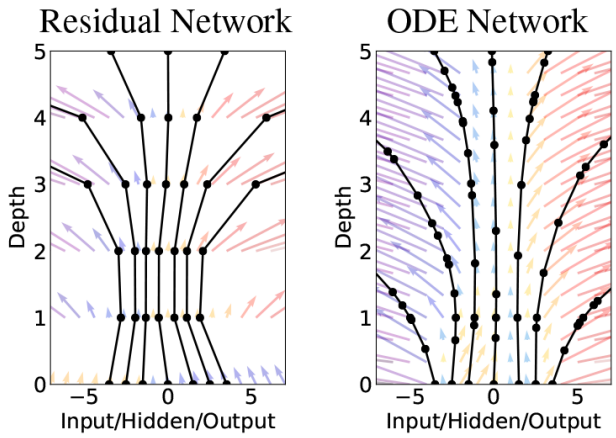


Figure 6: A residual network defines a series of discrete steps while an ODE network defines a continuous vector field [3].

3.2 The General Set-Up

A more or less intuitive way to understand how neural ODEs naturally extend the traditional residual network set-up, is that neural ODEs have infinitely many layers at different point in time. As such, we are interested in optimizing the trajectory, or the direction of change, of how the input maps to the output in the time span of the model. Instead of different weights and biases at each layer, we use a set of parameters to compute the derivative of the ultimate function we want to approximate. In other words, given the task of modeling the mapping of x to y in the space R^n, we are interested in optimizing the derivative such that the solution to the the initial value problem

$$\frac{dz}{dt} = f(z, \theta, t) \tag{30}$$

$$z(t_0) = x \tag{31}$$

will accurately predict y at time t_1.

The set-up of a neural ODE therefore require three basic ingredients: a derivative model f to compute the dynamics at a given time t, a set of parameter θ to calculate such a model, and a time span $[t_0, t_1]$ to evaluate the network. The continuous analog to matrix multiplication and linear algebra in evaluating traditional neural networks is then to integrate the derivative model over its time span:

$$z(t_1) = z(t_0) + \int_{t_0}^{t_1} f(z, \theta, t)dt. \tag{32}$$

.

This can be done fairly efficiently using modern ODE solvers, with the benefit that many such solvers are already available across different systems and extremely well tested. Given an input, we simply make a call to the ODE solver to evaluate the integral, plugging in the necessary initial value, derivative function, parameters and the time span:

$$z(t_1) = ODESolve(z(t_0), f, \theta(t), t_0, t_1). \tag{33}$$

Also similar to the standard neural network approach, the ultimate objective is for $z(t_1)$ to get as close to the desired labeled output y as possible. Thus, we also need a cost function to assess the performance of neural ODEs at each iteration. We define an arbitrary Cost function that takes in the integral output at time t_1:

$$\text{Cost}(z(t_1)) = \text{Cost}\left(z(t_0) + \int_{t_0}^{t_1} f(z(t), \theta(t), t)dt\right) \tag{34}$$

$$= \text{Cost}(ODESolve(z(t_0), f, \theta(t), t_0, t_1)). \tag{35}$$

It is not immediately obvious, however, how we would be able to optimize our neural ODE model. With traditional neural network, we compute the gradients with respect to the weights and biases at each layer and adjust them accordingly to achieve better results with gradient descent. To implement the same method for neural ODEs, we would then need to calculate the gradients with respect to their ingredients, namely the parameters θ as well as the start and stop time t_0, t_1. Furthermore, in order for neural ODE to work as a layer in a larger network, we would also want to compute the gradient with respect to the input $z(t_0)$. While such a task might be straight-forward in traditional networks, evaluating the gradients for neural ODEs would require us to differentiate under the integral sign without knowing the explicit formula of z in terms of θ and t.

Consequently, we introduce the Adjoint method [10, 3] in the next section, which can act as the reverse-mode differentiation of an ODE solution. Back propagation then will be fairly identical to our set-up of traditional neural networks, and different schemes of gradient descent will also be applicable.

3.3 Adjoint Method

Similar to a standard neural network, we wish to optimize the parameters of the neural ODE in order to minimize the Cost function. The main obstacle, however, is that back-propagating through an ODE solver is difficult and vastly inefficient. Moreover, it is also not obvious how we might compute the gradient of the Cost function with respect to θ without knowing an explicit solution of the state z in terms of θ and t. This is the motivation for the adjoint method to calculate the gradient without knowing the explicit solution.

Before we take a closer look at the adjoint method, a significant assumption we're undertaking in this section is that $f(z, \theta, t)$ together with its partial derivatives with respect to z, θ and t are continuous within the ranges of the variables. This is a fair assumption, since in practice we're modeling $f(z, \theta, t)$ using linear algebra computations. In addition, we also assume that θ does not change according to time. While setting θ to be a function of time might improve training, it is outside the scope of the adjoint method and other sensitivity

analysis techniques are preferred. As such, the adjoint method is only a computationally cheap and fast way to back propagate through specific classes of neural ODEs.

Note that $\frac{dz_t}{dt} = f(z_t, \theta, t)$, we first introduce a *Lagrangian* $\mathcal{L}$ function as follows

$$\mathcal{L} = \text{Cost}(z_{t_N}) - \int_{t_0}^{t_N} \lambda \left(\frac{dz}{dt} - f(z, \theta, t) \right) dt, \tag{36}$$

where t_0 and t_N compose the time span of our neural ODE, and λ is called the *Lagrange multiplier*, an arbitrary row vector chosen at time t.

The insight here is that the second term to the right is 0 by construction, and thus $\mathcal{L}$ would be the same as the Cost function. A clever choice of λ, however, will allow us to calculate $\frac{d\mathcal{L}}{d\theta}$ without solving for the explicit derivative of z with respect to θ. For convenience, the gradients of the Cost are assumed to be row vectors.

We first simplify $\mathcal{L}$ by integrating by parts:

$$\int_{t_0}^{t_N} \lambda \frac{dz}{dt} = \lambda z \Big|_{t_0}^{t_N} - \int_{t_0}^{t_N} \dot{\lambda} z = \lambda(t_N) z_{t_N} - \lambda(t_0) z_{t_0} - \int_{t_0}^{t_N} \dot{\lambda} z dt. \tag{37}$$

Substituting equation (37) into (36) results in

$$\mathcal{L} = \text{Cost}(z_{t_N}) - \lambda(t_N) z_{t_N} + \lambda(t_0) z_{t_0} + \int_{t_0}^{t_N} \dot{\lambda} z dt + \int_{t_0}^{t_N} \lambda f(z, \theta, t) dt. \tag{38}$$

Recall that λ is an arbitrary chosen at time t and so does not depend on θ. Additionally, z_{t_0} does not depend on θ either. We then derive $\mathcal{L}$ with respect to θ. Since f and its partial derivatives are continuous, we apply Leibniz's differentiation rule under the integral sign [11, Chapter 8]:

$$\frac{d\mathcal{L}}{d\theta} = \frac{d\text{Cost}}{dz_{t_N}} \frac{dz_{t_N}}{d\theta} - \lambda(t_N) \frac{dz_{t_N}}{d\theta} + \int_{t_0}^{t_N} \dot{\lambda} \frac{dz}{d\theta} dt + \int_{t_0}^{t_N} \lambda \left(\frac{\partial f}{\partial z} \frac{dz}{d\theta} + \frac{\partial f}{\partial \theta} \right) dt \tag{39}$$

$$= \left(\frac{d\text{Cost}}{dz_{t_N}} - \lambda(t_N) \right) \frac{dz_{t_N}}{d\theta} + \int_{t_0}^{t_N} \left(\dot{\lambda} + \lambda \frac{\partial f}{\partial z} \right) \frac{dz}{d\theta} dt + \int_{t_0}^{t_N} \lambda \frac{\partial f}{\partial \theta} dt. \tag{40}$$

Equation (40) motivates us to eliminate the first two terms. Indeed, we come up with a scheme to choose λ as follows:

$$\lambda(t_N) = \frac{d\text{Cost}}{dz_{t_N}} \tag{41}$$

$$\dot{\lambda} = -\lambda \frac{\partial f}{\partial z}. \tag{42}$$

Lemma 3.1. *Let us define the adjoint state $a(t)$ as the solution to the initial value problem*

$$a(t_N) = \frac{d\text{Cost}}{dz_{t_N}}, \qquad \frac{da}{dt} = -a(t) \frac{\partial f}{\partial z}, \tag{43}$$

then $a(t) = \frac{d\text{Cost}}{dz_t}$*, and the gradients of the* Cost *with respect to* z_{t_0}*,* θ*,* t_0 *could all be computed by evaluating the initial value problems at time* t_0*:*

$$\left.\frac{d\text{Cost}}{d\theta}\right|_{t_N} = a_\theta(t_0), \qquad a_\theta(t_N) = \mathbf{0}, \qquad \frac{da_\theta}{dt} = -a(t)\frac{\partial f}{\partial \theta} \tag{44}$$

$$\left.\frac{d\text{Cost}}{dt_0}\right|_{t_N} = -a_t(t_0), \qquad a_t(t_N) = -a(t_N)f(z_{t_N}, p, t_N), \qquad \frac{da_t}{dt} = -a(t)\frac{\partial f}{\partial t}. \tag{45}$$

Proof. Recall that $\mathcal{L} = \text{Cost}$, the chosen adjoint state now makes equation (40) much cleaner:

$$\left.\frac{d\text{Cost}}{d\theta}\right|_{t_N} = \frac{d\mathcal{L}}{d\theta} = \int_{t_0}^{t_N} \lambda\frac{\partial f}{\partial \theta}dt = \int_{t_0}^{t_N} a(t)\frac{\partial f}{\partial \theta}dt = \int_{t_N}^{t_0} -a(t)\frac{\partial f}{\partial \theta}dt = a_\theta(t_0) - a_\theta(t_N), \tag{46}$$

where a_θ is a function of t such that $\frac{da_\theta}{dt} = -a(t)\frac{\partial f}{\partial \theta}$.

To simplify the computation, we assume that $a_\theta(t_N) = \mathbf{0}$. Indeed, even if $a_\theta(t_N) \neq \mathbf{0}$, we simply choose $a'_\theta(t) = a_\theta(t) - a_\theta(t_N)$:

$$\left.\frac{d\text{Cost}}{d\theta}\right|_{t_N} = \int_{t_N}^{t_0} -a(t)\frac{\partial f}{\partial \theta} = a'_\theta(t_0) - a'_\theta(t_N) = a'_\theta(t_0). \tag{47}$$

This is the solution at time t_0 of the ODE (44). Compared to the original proof of the Neural ODE paper [3], equation (46) makes it clear that the gradient with respect to θ evaluated at time t_N is not $a_\theta(t_0)$, but $a_\theta(t_0) - a_\theta(t_N)$. The adjoint a_θ is only used to efficiently compute the gradient with respect to θ without having to know $\frac{dz}{d\theta}$. It does not represent the gradient with respect to θ evaluated at some time t. We therefore are free to assume $a_\theta(t_N) = \mathbf{0}$.

The same approach could be used to find the gradient with respect to the state z at any given point in time. Note that we use t' as a stand-in variable for t to disambiguate the lower bound t of the integral from the derivative variable. We define the Lagrangian at time t as:

$$\mathcal{L}(t) = \text{Cost}(z_{t_N}) - \int_t^{t_N} a(t')\left(\frac{dz}{dt'} - f(z, \theta, t')\right)dt' \tag{48}$$

$$= \text{Cost}(z_{t_N}) - \int_t^{t_N} a(t')\frac{dz}{dt'}dt' + \int_t^{t_N} a(t')f(z, \theta, t')dt' \tag{49}$$

$$= \text{Cost}(z_{t_N}) - a(t_N)z_{t_N} + a(t)z(t) + \int_t^{t_N} \frac{da}{dt'}zdt' + \int_t^{t_N} a(t')f(z, \theta, t')dt'. \tag{50}$$

Once again, we utilize Leibniz's differentiation rule under the integral sign [11, Chapter 8]:

$$\left.\frac{d\text{Cost}}{dz_t}\right|_{t_N} = \frac{d\mathcal{L}}{dz_t} \tag{51}$$

$$= \frac{d\text{Cost}}{dz_{t_N}}\frac{dz_{t_N}}{d_{z_t}} - a(t_N)\frac{dz_{t_N}}{dz_t} + a(t) + \int_t^{t_N} \frac{da}{dt'}\frac{dz}{dz_t}dt' + \int_t^{t_N} a(t')\frac{\partial f}{\partial z}\frac{dz}{dz_t}dt' \tag{52}$$

$$= \left(\frac{d\text{Cost}}{dz_{t_N}} - a(t_N)\right)\frac{dz_{t_N}}{dz_t} + a(t) + \int_t^{t_N} \left(\frac{da}{dt'} + a(t')\frac{\partial f}{\partial z}\right)\frac{dz}{dz_t}dt' \tag{53}$$

$$= a(t), \tag{54}$$

completing our proof that the adjoint state at time t is indeed the gradient of the Cost function with respect to the hidden state at t. The gradient of the Cost with respect to z at time t_0 is therefore simple $a(t_0)$.

Finally, we calculate the gradient of the Cost with respect to a given time t via the chain rule:

$$\left.\frac{d\text{Cost}}{dt}\right|_{t_N} = \left.\frac{d\text{Cost}}{dz_t}\right|_{t_N} \frac{dz_t}{dt} = a(t)f(z,\theta,t). \tag{55}$$

While equation (55) can be used to calculate the the gradient with respect to a given time t, we will formulate it into an initial value problem. This allows us to make a single call to the ODE solver and calculate all necessary gradients at once. Deriving equation (55) gives:

$$\frac{d}{dt}\left(\left.\frac{d\text{Cost}}{dt}\right|_{t_N}\right) = \frac{da(t)}{dt}f(z,\theta,t) + a(t)\left(\frac{\partial f}{\partial z}\frac{dz}{dt} + \frac{\partial f}{\partial t}\right) \tag{56}$$

$$= -a(t)\frac{\partial f}{\partial z}f(z,\theta,t) + a(t)\frac{\partial f}{\partial z}\frac{dz}{dt} + a(t)\frac{\partial f}{\partial t} \tag{57}$$

$$= a(t)\frac{\partial f}{\partial t}. \tag{58}$$

The gradient with respect to time t_0 can then be calculated with the formula

$$\left.\frac{d\text{Cost}}{dt_0}\right|_{t_N} = \left.\frac{d\text{Cost}}{dt_N}\right|_{t_N} + \int_{t_N}^{t_0} a(t)\frac{\partial f}{\partial t}dt. \tag{59}$$

Let $a_t(t) = -\frac{d\text{Cost}}{dt}|_{t_N}$, then we confirm that the gradient with respect to time t_0 can be calculated by solving (45). The reason why we set a_t to be the negative inverse of the gradient is that we would then be able to compute $-a(t)\frac{\partial f}{\partial z}$, $-a(t)\frac{\partial f}{\partial \theta}$, and $-a(t)\frac{\partial f}{\partial t}$ efficiently by computing the dot product of $-a(t)$ with the Jacobian matrix computed from $f(z,\theta,t)$. □

Thus, the full algorithm for the adjoint method is to solve the initial value problems detailed in lemma 3.1 backward in time from t_1 to t_0. For efficiency, we concatenate all three adjoints for z, θ, and t into a single augmented adjoint s. The dynamics of a, a_θ, a_t, and hence s is computed as the dot product of $-a(t)$ and the Jacobian matrix of $f(z,\theta,t)$ with respect to z, θ, t. With only one call to the ODE solver on the augmented adjoint s, we are then able to calculate all necessary gradients with respect to $z(t_0)$, θ, t_0, and t_1.

3.4 Strengths and Limitations

While neural ODEs appear to be a natural extension of existing residual networks, and open up a new area of research integrated with the differential equations field, we point out that there are certain drawbacks as well as benefits to this approach.

An important implication of using readily available ODE solver suites is that neural ODE can take advantage of unprecedented tools for deep learning. Many modern ODE solvers, for example, offer options to choose relative and absolute tolerances, consequently neural ODEs

can potentially trade slower computing time for better accuracy [3]. Additionally, neural ODEs is able to adapt to computation needs as dynamics becomes increasingly complex without altering the model. Traditional deep learning approach usually adds more layers to improve training, but in the case of utilizing neural ODEs, the solver can simply increase the number of time steps. Many algorithms have already been developed and refined in the field of numerical computing for this purpose, allowing neural ODEs to smoothly integrate vast areas of knowledge of solving differential equations out of the box.

Many other mathematical benefits and technical merits have been discussed at lengths in the original paper [3]. However, much more recent research into neural ODEs has discovered substantial trade-offs. The continuous nature of neural ODEs, while desirable and suitable for continuous dynamics, implies that the topology of the input space is preserved [4]. In other words, the output has the same dimension and lives in the same space with the input. This makes neural ODEs especially susceptible to ill-formed problems where traditional neural networks succeed. To illustrate this point, we present a simple problem where neural ODEs fail to approximate the sine function.

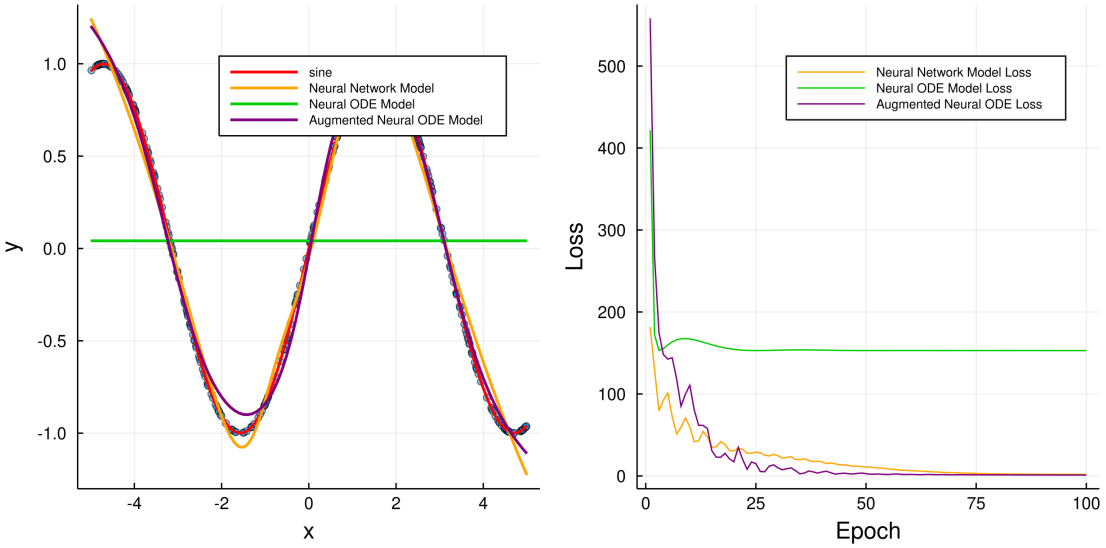


Figure 7: Approximating sine function. Neural network model is able to fit the sine function within the specified space relatively well, while a neural ODE model with similar set-up is stuck. The augmented neural ODE model is able to fit the sine wave within the specified region much better than a naive neural ODE network.

Figure 7 demonstrates a curve fitting problem, where both the neural ODE and the neural network models use only one hidden layer with eight nodes. We introduce some noise into the data, and compute the loss via sum of squared error from expected outputs. Both models run on the same data set for 100 epochs, but the neural ODE gets stuck early on, while the neural network is able to capture the sine wave within the region. The reason for the poor performance of the neural ODE is because sine is a function mapping from $\mathbb{R}$ to the interval $[-1, 1]$ where no vector field can replicate. Figure 8, for instance, shows that some trajectories must intersect each other with the start and stop time at 0 and 10, respectively.

This is particularly troubling for the naive neural ODE network with only one differential

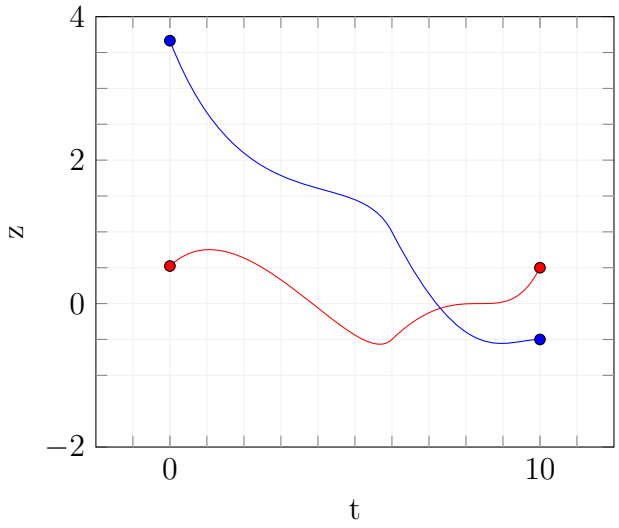


Figure 8: Continuous trajectories from $\frac{\pi}{6}$ to $\sin\frac{\pi}{6}$ and from $\frac{7\pi}{6}$ to $\sin\frac{7\pi}{6}$ must intersect and are not possible for neural ODE.

equation, since it is Lipschitz continuous [3]. According to Picard's existence and uniqueness theorem then, each point corresponds to an initial value problem and produces a unique solution [1]. All trajectories from the input to the output must therefore not cross each other. As we have shown, the sine function violates this assumption and so the neural ODE quickly fails. This limitation, however, could be avoided with the introduction of the augmented neural ODE.

3.5 Augmented Neural ODE

While there are classes of problems where a naive neural ODE will not perform favorably, we could extend the solution space of the network to sidestep this issue and improve training in general. The idea is that by adding extra dimensions, we are able to avoid clashing trajectories and ill-formed vector fields, thus accelerate training overall. In practice, we concatenate a few zero features to the input before pushing it to the ODE solver, and drop the additional dimensions from the output. This approach is aptly named Augmented Neural ODEs, and empirical experiments have shown that the model achieves better successes with less complex computations [4].

For instance, we update our last attempt at approximating the sine function by adding three extra dimensions to the input. Granted, there are many more parameters to tune and so better performance is to be expected. However, we note that the same number of parameters under the naive neural ODE network does not perform any better. Figure 7 demonstrates how training is enhanced after augmentation.

3.6 Mathematical Modeling

Since the essence of neural ODE is to learn continuous dynamics through time, it follows that a particularly fitting use case is to use neural ODE as an optimizable dynamical model

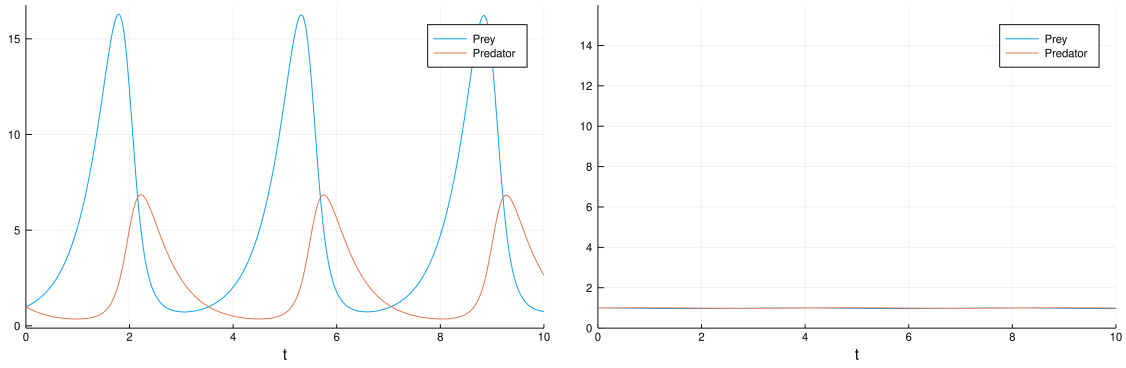
instead of a drop-in replacement for neural network. While deep learning has proven itself to be incredibly effective, it requires an an abundant amount of data for more complex tasks. Furthermore, the resulting model is not interpretable: testing confirms the accuracy of neural networks, but there is no immediate understanding of what weights and biases physically represent. A classification system might correctly identify an animal species from a photo, but we do not know what features have motivated this decision. On the other hand, mathematical models are highly interpretable, translating observations in reality into equations that can be easily understood. However, more often than not the formulation of such models has limited predictive power due to unrealistic simplifying assumptions. In this section, we explore how neural ODE could be used to model mathematical systems, and potentially combine mathematical modeling with deep learning to remain interpretable without sacrificing accuracy, with an example of the Lotka-Volterra predator-prey model.

The Lotka–Volterra model is a pair of first-order nonlinear differential equations that describe the population dynamics of a predator-prey biological system:

$$\frac{dx}{dt} = \alpha x - \beta xy \tag{60}$$

$$\frac{dy}{dt} = -\delta y - \gamma xy, \tag{61}$$

where x is the number of preys and y is the number of predators.



(a) Before Training (b) After Training

Figure 9: Lotka-Volterra Parameter Fitting with Neural ODE.

In the equations above, each derivative gives the instantaneous growth rate of the corresponding population. For instance, the population of the prey increases with the reproduction rate α, but decreases with more encounters between the preys and the predators. On the other hand, the predator population increases with more prey to hunt on, but decreases with more competition among its kind.

This example is of interest for two reasons. First, the solutions are periodic, but cannot be represented with simple trigonometric functions. Second, the model can be inserted directly into a neural ODE, and therefore is able to leverage gradient descent to find the appropriate parameters for a desired outcome. Figure 9 shows the result after training adapted from an example of Julia's DiffEqFlux library [13]. The objective is simply to have both populations

as close to 1 as possible, and the neural ODE is able to quickly detect the parameters satisfying this condition.

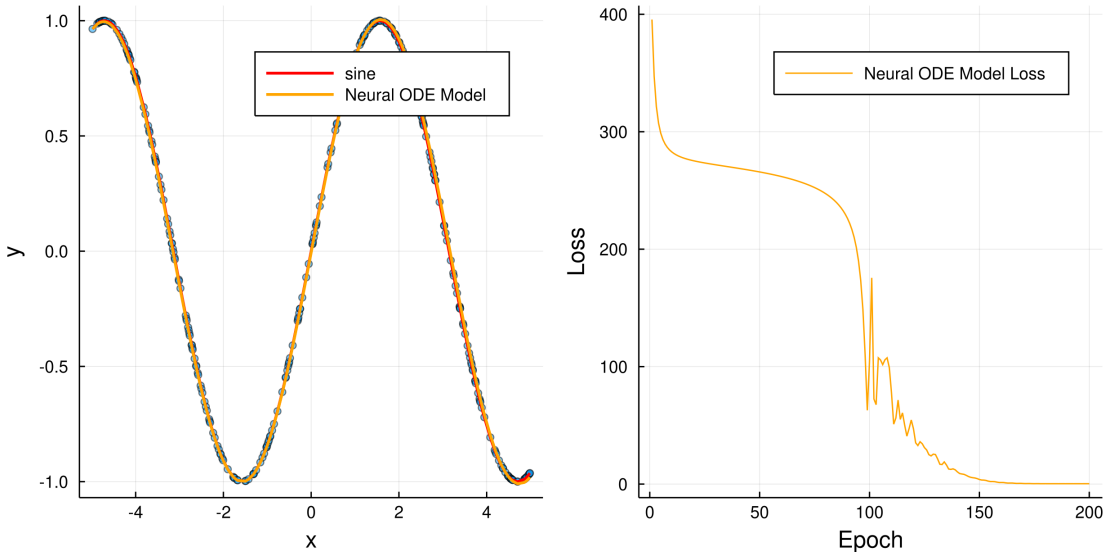


Figure 10: Neural ODE as an optimizable dynamical system. By approximating the derivative of the sine function on the data set, we are able to not only fit training data accurately but also extrapolate the correct dynamics for sine.

In fact, this approach could be used to approximate the sine function from the previous sections. We approximate the derivative at each point in the data set numerically, and build a dynamical model as follows:

$$\begin{bmatrix} y \\ dy \end{bmatrix} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix} \begin{bmatrix} y \\ dy \end{bmatrix}, \tag{62}$$

where α, β, γ, δ are all initiated to be 0. Since the derivative of the sine function is cosine, the correct parameters should be

$$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix},$$

which we successfully optimize in training. After 200 epochs, the neural ODE finds the optimal parameters to be $\alpha = 0.03345$, $\beta = 1.003$, $\gamma = -0.9876$, $\delta = -0.03052$. Unsurprisingly, the model fits the sine curve perfectly, and is able to extrapolate over the entire R set accurately. The full code can be found in Appendix B2.

We would like to note that this showcases how mathematical models can be smoothly integrated with neural ODE networks. On a broader scale, we will be able to impute prior mathematical knowledge into a deep learning model through neural ODE by providing the terms of the equations we already know, and use a neural network to approximate the remaining portions. Recent research into this direction have already achieved substantial results: using a combination of mathematical modeling and neural ODE allows for effective learning on tiny amount of data, and has the potential to automate discovery of explicit dynamics equations[12].

3.7 Neural ODE Implementation in Julia

An implementation of neural ODE networks and required libraries can be found in Appendix B1. The code for approximating the sine function is in Appendix B2, and the Lotka-Volterra system is modelled is in B3.

4 Conclusion

In this paper we lay the foundational framework for traditional neural networks and neural ODEs from a rigorous mathematical perspective. In addition, we demonstrate the motivation behind extending residual neural networks into neural ODEs, and analyze advantages as well as trade-offs of this new approach. We also show how augmentation might help enhance neural ODEs training, and explore how neural ODEs can be utilized for mathematical modeling. One potential direction for future research is then to further integrate neural ODEs with scientific and mathematical knowledge to automate interpretable formulation of hidden dynamics.

References

[1] R. P. Agarwal and V. Lakshmikantham. *Uniqueness and nonuniqueness criteria for ordinary differential equations*. World Scientific Publishing, 1993.

[2] Jeff Bezanson et al. "Julia: A fresh approach to numerical computing". In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: 10.1137/141000671.

[3] Tian Qi Chen et al. "Neural ordinary differential equations". In: *Advances in neural information processing systems*. 2018, pp. 6571–6583.

[4] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. *Augmented Neural ODEs*. 2019. arXiv: 1904.01681 [stat.ML].

[5] FluxML. *Flux.jl*. Version 0.10.4. URL: https://github.com/FluxML/Flux.jl.

[6] FluxML. *Zygote.jl*. Version 0.4. URL: https://github.com/FluxML/Zygote.jl.

[7] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[8] Catherine F Higham and Desmond J Higham. "Deep learning: An introduction for applied mathematicians". In: *SIAM Review* 61.4 (2019), pp. 860–891.

[9] Yiping Lu et al. "Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations". In: *arXiv preprint arXiv:1710.10121* (2017).

[10] Lev Semenovich Pontryagin. *Mathematical theory of optimal processes*. Routledge, 2018.

[11] Murray H. Protter and Charles B. Jr. Morrey. *Intermediate Calculus*. Springer Science Business Media, 2012.

[12] Christopher Rackauckas et al. *Universal Differential Equations for Scientific Machine Learning*. 2020. arXiv: 2001.04385 [cs.LG].

[13] Chris Rackauckas et al. "Diffeqflux. jl-A julia library for neural differential equations". In: *arXiv preprint arXiv:1902.02376* (2019).

[14] Oxford Reference. *Taylor's Theorem*. URL: https://www.oxfordreference.com/view/10.1093/oi/authority.20110803102744753.

[15] SciML. *DifferentialEquations.jl*. Version 6.9. URL: https://github.com/SciML/DifferentialEquations.jl.

Appendix: Julia Implementations

Appendix A1: Neural Network Implementation

We implement a neural network framework completely from scratch in Julia, using only native functions of the language without external libraries. This implementation supports arbitrary number of layers, is able to work with any activation and cost function as long as the derivative function is also defined, and can handle mini-batching. An example use case to generate the graphs for Section 1 in this paper can be found in Appendix A2.

```
### ============================================================================
### Layer Struct
### ============================================================================

# A Layer consists of weights, bias, and an activation function to produce the
# output given the input. Additionally, we also save the result before applying
# the activation function to compute the backward pass.
struct Layer{WS, BS, Z, F}
    W::WS    # weights
    b::BS    # biases
    z::Z     # intermediate state
    σ::F     # activation function
end

# The Layer constructor takes in the dimensions of the input, output, and an
# activation function. Weight and biases are set accordingly, and an empty array
# is set up to store the intermediate state.
Layer(in::Int, out::Int, σ::Function) =
    Layer(rand(Float32, out, in) .- 0.5f0,    # weights
          zeros(Float32, out),                # biases
          Array{Float32}[],                   # intermediate state vector
          σ)                                  # activation function

# Layer output is computed using the formula σ(W * X + b)
function (l::Layer)(X)
    W, b, z, σ = l.W, l.b, l.z, l.σ
    temp = W * X .+ b    # .+ broadcasting is also compatible with batches
    empty!(z)
    push!(z, temp)    # store intermediate state for back propagation
    return σ.(temp)    # apply the activation function element-wise
end

# Layer is updated with partial derivatives and learning rate.
function update!(l::Layer, dW, db, η)
    l.W .-= η * dW
    l.b .-= η * db
end
```

```
# Given the derivative of the Cost wrt the ouput, we calculate the partial
# derivatives with respect to weights, biases, and the input.
function derive(l::Layer, ∂Cost∂a_out, a_in)
    dσ = derive(l.σ)    # user-defined derivative function of σ

    # Cost wrt intermediate result
    ∂Cost∂z = ∂Cost∂a_out .* dσ.(l.z[1])

    # ∂W computes Cost wrt weights for one pair of input and output
    ∂W(∂Cost∂z, a_in) = ∂Cost∂z * a_in'
    # Cost wrt weights for entire batch
    ∂Cost∂W = sum(∂W.(eachcol(∂Cost∂z), eachcol(a_in)))

    # Cost wrt to bias
    ∂Cost∂b = sum(eachcol(∂Cost∂z))

    # Cost wrt input from last layer
    ∂Cost∂a_in = l.W' * ∂Cost∂z

    return ∂Cost∂W, ∂Cost∂b, ∂Cost∂a_in
end

# Back propagation given the input from the previous layer
# and the cost gradient wrt this layer's output
function back_propagate!(l::Layer, ∂Cost∂a_out, a_in, η)
    ∂Cost∂W, ∂Cost∂b, ∂Cost∂a_in = derive(l, ∂Cost∂a_out, a_in)    # gradients
    update!(l, ∂Cost∂W, ∂Cost∂b, η)    # update parameters
    return ∂Cost∂a_in    # Cost wrt input from last layer
end

### ============================================================================
### Model Struct
### ============================================================================

# A Model consists of multiple Layers. Additionally, we store each Layer's
# outputs in an array for the backward pass.
struct Model{LS, OS}
    layers::LS    # Layers
    a::OS         # Layer outputs

    # Model Constructor
    Model(layers...) = new{typeof(layers), Vector{Array{Float32}}}(layers, [])
end

# Evaluate Model by evaluating the layers sequentially.
function (m::Model)(X)
    # Store Model input
    empty!(m.a)
    push!(m.a, X)

    # Evaluate each layer and store their outputs
    for layer in m.layers
        push!(m.a, layer(m.a[end]))
    end

    # Return Model output
    return pop!(m.a)
end

# Back-propagate through the Model by back-propagating through each layer.
function back_propagate!(m::Model, ∂Cost∂aL, η)
    # Back propagate through each layer
    ∂Cost∂a_out = ∂Cost∂aL
    for layer in reverse(m.layers)
        a_in = pop!(m.a)    # retrieve layer input
        ∂Cost∂a_out = back_propagate!(layer, ∂Cost∂a_out, a_in, η)
    end
```

```
end

# Training requires a Model, a Cost function, the training dataset, and the
# learning rate.
function train!(m::Model, Cost, dataset, η)
    costs = Float32[]    # store cost of each batch in dataset
    dCost = derive(Cost)    # user-defined derivative function of Cost

    # Train Model on each batch in dataset
    for batch in dataset
        X, Y = batch
        out = m(X)

        # Calculate cost
        cost = Cost(out, Y)
        push!(costs, cost)

        # Back propagation
        ∂Cost∂out = dCost(out, Y)
        back_propagate!(m, ∂Cost∂out, η)
    end

    # Return average cost of all batches
    return sum(costs) / length(dataset)
end
```

Appendix A2: Neural Network Example

We use a simple classification example to demonstrate how neural networks are able to approximate and extrapolate hidden mappings by training on labeled data. The neural network only requires one hidden layer of 10 nodes to accurately approximate the underlying classification model.

```
### ============================================================================
### Load Packages
### ============================================================================

using Plots, LinearAlgebra
import Random
include("./NeuralNetwork.jl")

### ============================================================================
### Generate Training Data
### ============================================================================

# True model to classify data
model(x) = x^3 - 3x + 2

# Generate a data set given the classification model, the range of coordinates,
# and the size of the data set.
function generate_dataset(model, xrange, yrange, size)
    # Random sampling function
    sample(range, size) = rand(size) * (range[end] - range[1]) .+ range[1]

    # Sample x and y coordinates
    xs = sample(xrange, size)
    ys = sample(yrange, size)

    # Input matrix
    input = hcat(x1s, x2s)'
```

```
    # Points above and below the model are labeled 1 and 0, respectively
    output = Float32.(x2s .≥ model.(x1s))'

    return input, output
end

# Coordinate range of data set
x1grid = -3:0.1:3
x2grid = -16:0.1:20

# Generate the inputs and corresponding outputs
input, output = generate_dataset(model, x1grid, x2grid, 500)

# Plot the data set
ishigher = vec(output .== 1)
function plot_data()
    scatter(input[1, ishigher], input[2, ishigher], label="", color=:blue)
    scatter!(input[1, .!ishigher], input[2, .!ishigher], label="", color=:red)
    plot!(x1grid, model.(x1grid), label="True Model", legend=:outertopright)
end
plot_data()

### ============================================================================
### Training Environment
### ============================================================================

### Activation functions
### ====================
# Sigmoid activation function and its derivative
sigmoid(x) = 1 / (1 + exp(-x))
derive(::typeof(sigmoid)) = x -> sigmoid(x) * (1 - sigmoid(x))

# Relu activation function and its derivative
relu(x) = max(0, x)
derive(::typeof(relu)) = x -> (x ≥ 0 ? 1 : 0)

### Neural network model
### ====================
neural_model = Model(
    Layer(2, 10, relu),
    Layer(10, 1, sigmoid))

### Cost objective
### ==============
# Binary cross entropy loss and its derivative
bcentropy(ŷ, y) = -y* log(ŷ + 1e-7) - (1 - y) * log(1 - ŷ + 1e-7)
derive(::typeof(bcentropy)) = (ŷ, y) -> -y/(ŷ + 2e-7) + (1 - y)/(1 - ŷ + 1e-7)

# Cost function and its derivative
Cost(Ŷ, Y) = sum(bcentropy.(Ŷ, Y)) / size(Y, 2)
derive(::typeof(Cost)) = (Ŷ, Y) -> derive(bcentropy).(Ŷ, Y) / size(Y, 2)

### Minibatching
### ============
function minibatch(input, output, batch_size)
    inputs = Array[]
    outputs = Array[]

    # Shuffle data set and split into batches
    rand_idxs = Random.randperm(size(input, 2))
    batch_idxs = Iterators.partition(rand_idxs, batch_size)
    for batch_idx in batch_idxs
        push!(inputs, input[:, batch_idx])
        push!(outputs, output[:, batch_idx])
    end

    # Return data set
```

```
    zip(inputs, outputs) |> collect
end

### Model visualization
### ====================
function plot_model(neural_model, x1grid, x2grid)
    plot_data()    # plot true model

    # Find the classification boundary
    bound = fill(maximum(x2grid), size(x1grid))
    for (i, x1) in enumerate(x1grid)
        for x2 in x2grid
            out = neural_model([x1, x2])[end]
            if out ≥ 0.5
                bound[i] = x2
                break
            end
         end
    end

    # Plot neural network model
    plot!(x1grid, bound, label="Neural Network Model")
end

### ================================================================================
### Training
### ================================================================================

η = 0.05    # learning rate
epochs = 1000    # training epochs
anim = Animation()    # visualization of training process

### Training loop
### =============
for i in 1:epochs
    # Prepare batches
    dataset = minibatch(input, output, 5)

    # Train model
    cost = train!(neural_model, Cost, dataset, η)

    # Report loss
    println("Epoch $i average cost: $cost")

    # Update visualization
    if i % 10 == 0
        plot_model(neural_model, x1grid, x2grid)
        frame(anim)
    end
end

### Trained model visualization
### ===========================
gif(anim)
plot_model(neural_model, x1grid, x2grid)
```

Appendix B1: Neural ODE Implementation

We implement a neural ODE framework with Flux, a machine learning package, Zygote, a source to source automatic differentiation package, and DifferentialEquations, an ODE solver in Julia.

```julia
using DifferentialEquations, Flux, Zygote

# Extend Zygote to work with Neural ODE
function Zygote._zero(xs::AbstractArray{<:AbstractArray}, T=Any)
    return [Zygote._zero(x) for x in xs]
end

### ============================================================================
### Neural ODE Layer Struct and Constructor
### ============================================================================

# A Neural ODE consists of a function f(z, θ, t) that models z's derivative, and
# the parameters `θ` to be optimized, and the time span of the integral.
# Additionally, we also store the ODE solver's solution, since it is useful for
# the backward pass via the adjoint method.
struct NeuralODE{F, P, T, S}
    f::F     # derivative model
    θ::P     # vector of parameters
    tspan::T # time span [t0, t1]
    sol::S   # vector of ODE solution
end

# We store the ODE solver's solution in a vector instead of directly to make the
# `NeuralODE` struct immutable for better performace. At initialization, this is
# simply an empty vector.
function NeuralODE(f, θ, tspan)
    return NeuralODE(f, θ, tspan, DiffEqBase.AbstractODESolution[])
end

### ============================================================================
### Flux compatibility
### ============================================================================

# Using the macro `Flux.@functor` allows the machine learning library Flux to
# mix our `NeuralODE` layer in any model.
Flux.@functor NeuralODE

# We also specify the parameters `θ` to be optimized with `Flux.trainable`. We
# only update `θ` by default, but we can also optimize the time span.
Flux.trainable(node::NeuralODE) = (node.θ,)

### ============================================================================
### Forward pass
### ============================================================================

# The forward pass computes the integration with the ODE solver. The forward
# pass returns an array of the solution at each timestep.
function (node::NeuralODE)(z_t0; alg=Tsit5(), kwargs...)
    f, θ, t0, t1, sol = node.f, node.θ, node.tspan[1], node.tspan[2], node.sol
    return forward!(z_t0, θ, t0, t1; f=f, sol=sol, alg=alg, kwargs...)
end

# Integrate from `t0` to `t1` to calculate `z` at `t1`, also returns `z` at
# each timestep in a vector.
function forward!(z_t0, θ, t0, t1; f, sol, alg, kwargs...)
    # Define and solve ODE problem
    function dzdt(dz, z, θ, t)
        dz .= f(z, θ, t)
    end
    problem = ODEProblem(dzdt, z_t0, (t0, t1), θ)
    solution = solve(problem, alg; kwargs...)

    # Store the solution for the backward pass
    empty!(sol)
    push!(sol, solution)

    # Return an array of `z` evaluated at each timestep
    return solution.u
```

```
end

### ================================================================================
### Backward pass
### ================================================================================

# Since back-propagating through the ODE solver is complex, we define a custom
# backward pass for the Neural ODE via the adjoint method. Flux relies on the
# Zygote library to calculate gradients, and we can define our custom gradient
# via `Zygote.@adjoint`.
Zygote.@adjoint function forward!(z_t0, θ, t0, t1; f, sol, alg, kwargs...)
    # Forward pass
    zs = forward!(z_t0, θ, t0, t1; f=f, sol=sol, alg=alg, kwargs...)

    # Return the forward pass and how to calculate the gradients of the loss wrt
    # `z_t0` and `θ` from the gradient of the loss wrt `z` at each timestep.
    return zs, ∂L∂zs -> backward(∂L∂zs, θ; f=f, sol=sol[1], alg=alg)
end

# Compute the gradients of the loss wrt to `θ`.
function backward(∂L∂zs, θ; f, sol, alg)
    # Calculate the partial derivatives from each relevant `∂L∂z`
    idxs = .!(iszero.(∂L∂zs)) |> collect
    t0 = sol.t[1]
    t1s = sol.t[idxs]
    ∂s = _backward.(∂L∂zs[idxs], Ref(θ), t0, t1s; f=f, sol=sol, alg=alg)

    # Aggregate all partial derivatives
    ∂L∂t1 = ∂s[end][end]
    ∇ = map(+, [∂[1:3] for ∂ in ∂s]...)
    return (∇..., ∂L∂t1)
end

# Given the gradient of the loss wrt `z` at time `t1`, compute the partial
# derivatives wrt `z_t0` and `θ` via the adjoint method.
function _backward(∂L∂z_t1, θ, t0, t1; f, sol, alg)
    # Derivative of the loss wrt `t1`
    ∂L∂t1 = ∂L∂z_t1[:]' * f(sol[end], θ, t1)[:]

    # We define the initial augmented state, which consists of the gradients of
    # the loss wrt to `z_t1` and `θ` and `t1`. `ArrayPartition` from the
    # DifferentialEquations library allows us to combine arrays with different
    # dimensions for a single call to the ODE solver.
    s_t1 = ArrayPartition(∂L∂z_t1, zero(θ), [-∂L∂t1])

    # Define the dynamics of the augmented state
    function dsdt(ds, s, θ, t)
        # Compute the Jacobian matrices of `f` wrt `z`, `θ`, and `t`
        _, back = Zygote.pullback(f, sol(t), θ, t)

        # Adjoint dynamics
        d = back(-s.x[1])

        # Zygote returns `nothing` as a strong zero if the function is not
        # dependent on the variable, so we convert to zero for computation
        get_derivative(Δ, x) = (Δ == nothing ? zero(x) : Δ)
        Δs = get_derivative.(d, ds.x[:])

        # Return the derivatives
        for i in 1:3
            ds.x[i] .= Δs[i]
        end
    end

    # Solve ODE backwards
    problem = ODEProblem(dsdt, s_t1, (t1, t0), θ)
    solution = solve(problem, alg)
    s_t0 = solution[end]
```

```
    # Return gradients
    return (s_t0.x[1], s_t0.x[2], -s_t0.x[3][1], ∂L∂t1)
end
```

Appendix B2: Neural ODE Sine Approximation Example

We use a simple example of approximating the sine function to show how neural ODEs can get stuck in ill-formed solution space, while traditional neural networks are still able to perform well. Furthermore, we demonstrate how augmented neural ODEs can side-step this issue, and enhance training in general.

```
### ================================================================================
### Load Packages
### ================================================================================

include("./NeuralODE.jl")
using Plots

### ================================================================================
### Generate Training Data
### ================================================================================

# Random input from -5 to 5
X = rand(Float32, 300) * 10 .- 5 |> sort!

# Sine function output with added noise
Y = sin.(X) .+ Float32(1e-3) * (rand(Float32, 300) .- 0.5f0)

# Visualization of sine function and training data
xgrid = -5:0.1:5
function plot_data()
    scatter(X, Y, label=:none, ms=3, alpha=0.5)    # training data
    plot!(xgrid, sin.(xgrid), label="sine", lw=2, c=:red)    # sine function
end
pl1 = plot_data()

### ================================================================================
### Neural Network Model
### ================================================================================

# Define neural network model
nn_model = Chain(Dense(1, 8, tanh), Dense(8, 1))

# Parameters to be optimized
nn_params = Flux.params(nn_model)

# Sum of squared error as loss function
nn_loss() = sum(abs2, nn_model(reshape(X, 1, :)) - Y')

# Set up to run for 100 epochs
nn_data = Iterators.repeated((), 100)

# Optimizer
nn_opt = ADAM(0.1)

# Store loss each epoch for visualization
nn_losses = Float32[]
nn_cb = () -> begin
    push!(nn_losses, nn_loss())
end
```

```
# Training loop
Flux.train!(nn_loss, nn_params, nn_data, nn_opt, cb=nn_cb)

# Plot losses versus epoch
pl2 = plot(1:100, nn_losses, label="Neural Network Model Loss",
     xlabel="Epoch", ylabel="Loss", c=:orange)

# Network Visualization
plot!(pl1, xgrid, nn_model(xgrid')', c=:orange, lw=2,
      xlabel="x", ylabel="y", label="Neural Network Model")

### ============================================================================
### Naive Neural ODE Model
### ============================================================================

# Derivative model
model = Chain(Dense(1, 8, tanh), Dense(8, 1))
θ, re = Flux.destructure(model)
dzdt(z, θ, t) = re(θ)(z)

# Define Neural ODE
node_model = NeuralODE(dzdt, θ, [0.0f0, 10.0f0])

# Parameters to be optimized, including time span
Flux.trainable(node::NeuralODE) = (node.θ, node.tspan)
node_params = Flux.params(node_model)

# Sum of squared error as loss function
node_loss() = sum(abs2, node_model(reshape(X, 1, :))[end] - Y')

# Set up to run for 100 epochs
node_data = Iterators.repeated((), 100)

# Optimizer
node_opt = ADAM(0.1)

# Store losses for visualization
node_losses = Float32[]
node_cb = () -> begin
    push!(node_losses, node_loss())
end

# Training loop
Flux.train!(node_loss, node_params, node_data, node_opt, cb=node_cb)

# Plot losses versus epochs
plot!(pl2, 1:100, node_losses, label="Neural ODE Model Loss",
      c=:green3, xlabel="Epoch", ylabel="Loss")

# Neural network visualization
plot!(pl1, xgrid, node_model(xgrid')[end]', lw=2,
      xlabel="x", ylabel="y", label="Neural ODE Model", c=:green3)

### ============================================================================
### Augmented Neural ODE Model
### ============================================================================

# Derivative model
model = Chain(Dense(4, 8, tanh), Dense(8, 4))
θ, re = Flux.destructure(model)
dzdt(z, θ, t) = re(θ)(z)

# Define Augmented Neural ODE
anode_model = NeuralODE(dzdt, θ, [0.0f0, 10.0f0])

# Parameters to be optimized, including time span
anode_params = Flux.params(anode_model)
```

```julia
# Input augmentation
aug_X = hcat(X, zeros(eltype(X), length(X), 3)) |> transpose

# Sum of squared error as loss function
anode_loss() = sum(abs2, anode_model(aug_X)[end][1, :] - Y)

# Set up to run for 100 epochs
anode_data = Iterators.repeated((), 100)

# Optimizer
anode_opt = ADAM(0.01)

# Store losses for visualization
anode_losses = Float32[]
anode_cb = () -> begin
    push!(anode_losses, anode_loss())
end

# Training loop
Flux.train!(anode_loss, anode_params, anode_data, anode_opt, cb=anode_cb)

# Plot losses versus epochs
plot!(pl1, 1:100, anode_losses, label="Augmented Neural ODE Loss",
      xlabel="Epoch", ylabel="Loss", c=:purple)

# Augmented Neural ODE visualization
aug_xgrid = hcat(xgrid, zeros(eltype(xgrid), length(xgrid), 3)) |> transpose
plot!(pl2, xgrid, anode_model(aug_xgrid)[end][1, :], lw=2, c=:purple,
      xlabel="x", ylabel="y", label="Augmented Neural ODE Model")

# Plot all visualizations
plot(pl1, pl2, legendfontsize=6, size=(800, 400))

### ================================================================================
### Mathematical Neural ODE Model
### ================================================================================

# Derivative model for [Y, dY]
θ = zeros(Float32, 2, 2)
dzdt(z, θ, t) = θ * z

# Define ODE
ode_model = NeuralODE(dzdt, θ, [X[1], X[end]])

# Parameters to be optimized
Flux.trainable(ode::NeuralODE) = (ode.θ,)
ode_params = Flux.params(ode_model)

# Set up to run for 200 loops
ode_data = Iterators.repeated((), 200)

# Approximation of first derivative from data set
dY = [(Y[i+1] - Y[i-1])/(X[i+1] - X[i-1]) for i in 2:length(X)-1]
pushfirst!(dY, (Y[2] - Y[1])/(X[2] - X[1]))
push!(dY, (Y[end] - Y[end-1])/(X[end] - X[end-1]))

# Solve for all solutions at each time step X
predict() = ode_model([Y[1], dY[1]], saveat=X)

# Sum of squared error as loss function
ode_loss() = begin
    predicted = predict()
    sum(abs2, hcat(predicted...) - [Y'; dY'])
end

# Optimizer
ode_opt = ADAM(0.1)
```

```julia
# Store losses for visualization
ode_losses = Float32[]
ode_cb = () -> begin
    push!(ode_losses, ode_loss())
end

# Training loop
Flux.train!(ode_loss, ode_params, ode_data, ode_opt, cb=ode_cb)

# Plot losses versus epochs
pl3 = plot(1:200, ode_losses, label="Neural ODE Model Loss",
           c=:orange, xlabel="Epoch", ylabel="Loss")

# Mathematical Neural ODE visualization
pl4 = plot_data()
Ŷ = (predicted = predict(); [p[1] for p in predicted])
plot!(pl4, X, Ŷ, lw=2, c=:orange,
      xlabel="x", ylabel="y", label="Neural ODE Model")

# Show plots together
plot(pl4, pl3, size=(800, 400))
```

Appendix B3: Neural ODE Lotka-Volterra Example

We show how neural ODEs can be integrated with mathematical modeling to optimize parameters for some objectives with the Lotka-Volterra example adapted from the DiffEqFlux's tutorial using our own neural ODE framework. The result demonstrates the potentials of neural ODEs in mathematical modeling out of the box. Future research can further look into how mathematical models can be integrated with deep learning to enhance interpretability of neural network models without sacrificing efficiency and effectiveness.

```julia
### ============================================================================
### Load Packages
### ============================================================================

include("./NeuralODE.jl")
using Plots

### ============================================================================
### Define and Solve Lotka-Volterra Model
### ============================================================================

# Lotka-Volterra equations
function model(z, θ, t)
    x, y = z
    α, β, δ, γ = θ
    return [α*x - β*x*y,
            -δ*y + γ*x*y]
end

# Starting populations at time 0
z0 = [1.0f0, 1.0f0]

# Parameters
θ = [2.2f0, 1.0f0, 2.0f0, 0.4f0]

# Define and solve ODE problem
problem = ODEProblem(model, z0, (0.0f0, 10.0f0), θ)
sol = solve(problem, Tsit5())
```

```julia
# Visualize solution
plot(sol)

### ================================================================================
### Neural ODE with Lotka-Volterra equations
### ================================================================================

# Define Neural ODE layer
node = NeuralODE(model, θ, [0.0f0, 10.0f0])

# Parameters to be optimized
params = Flux.params(node)

# Sum of squared error from 1 as loss function
function loss()
    zs = node(z0, saveat=0.1f0)
    return sum(abs2, vcat(zs...) .- 1.0f0)
end

# Set up the Neural ODE 100 to run for 100 epochs
data = Iterators.repeated((), 100)

# Optimizer
opt = ADAM(0.1)

# Update the plot of our populations each epoch
anim = Animation()
cb = () -> begin
    plot(solve(remake(problem; p=node.θ), Tsit5()),
         xlims=(0, 10), ylims=(0, 16))
    frame(anim)
end

# Training Loop
Flux.train!(loss, params, data, opt, cb=cb)

# Training visualization
gif(anim)
```